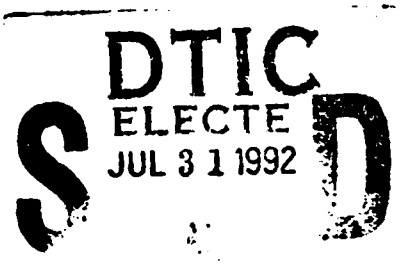
UNCLASSIFIED



AFIT/EN-TR-92-2

Air Force Institute of Technology

Object-Oriented Design Unifies Databases and Applications

Douglas E. Dyer Mark A. Roth

13 July 1992

Approved for public release; distribution unlimited

13 July 1992 Technical Report

Object-Oriented Design Unifies Databases and Applications

Douglas E. Dyer, Capt, USAF
Mark A. Roth, Maj, USAF

Air Force Institute of Technology, WPAFB OH 45433-6583

AFIT/EN-TR-92-2

ASC/RWWW
Wright-Patterson AFB OH, 45433-6503

Distribution Unlimited

An object-oriented design methodology, such as the object modeling technique (OMT) of Rumbaugh et al., not only unifies database design with applications software design, it abstracts the design process and supports implementation regardless of the data model used.

Object-Oriented Design, Database Design, Software Systems Modeling 27

UNCLASSIFIED UNCLASSIFIED UNCLASSIFIED UL

Object-Oriented Design Unifies Databases and Applications*

Douglas E. Dyer and Mark A. Roth

Abstract

An object-oriented design methodology, such as the object modeling technique (OMT) of Rumbaugh et al., not only unifies database design with applications software design, it abstracts the design process and supports implementation regardless of the data model used.

1 Introduction

Database design and design of software applications have previously been two separate activities. Increasingly, however, the object-oriented approach is unifying design techniques, just as it has begun to unify database and software technologies. Object-oriented design methods, such as the Object Modeling Technique (OMT) of Rumbaugh et al., combine structural models, normally used in database design, with dynamic and functional models, which are frequently used in software design. Therefore, OMT and other, similar, object-oriented design methodologies, provide a basis for both database design and application software design. Furthermore, OMT is applicable regardless of which data model is supported by the chosen database management system.

From our research, we have concluded that object-oriented design methods have rendered *separate* design phases for an object-oriented database and the object-oriented applications using it not only unnecessary, but undesirable. Instead, a single design phase should consider both at once. Furthermore, a unified object-oriented design methodology does not limit the choice of implementation language or database. In particular, traditional relational or network database structures may be extracted from the object-oriented design when there is no

*This research funded by Joint Modeling and Simulation Systems Program Office, ASC/RWWW, Wright-Patterson AFB, OH 45433

need for an object-oriented database, or when the risk of using this relatively new technology is considered too great.

This paper introduces OMT as an exemplar design methodology of the object-oriented viewpoint and explains how we have arrived at these conclusions.

2 Background

We have been studying design methodologies for object-oriented databases because of the increasing availability of database management systems having object-oriented features. A number of authors have addressed object-oriented design methods for application software development, especially for software written in a general-purpose object-oriented programming language. However, databases have characteristics which are distinct from normal software. Databases are relatively passive compared to procedural programs and databases differ from ordinary programs by retaining their data between program invocations. We felt that these kinds of differences would lead to different requirements in design methods.

We began our research by reviewing the literature for design methods intended specifically for use with object-oriented databases. Different 'object-oriented' DBMS have different capabilities and are based on different data models. At this time, there is really no characteristic set of features that can be expected of an 'object-oriented' DBMS.[1] Therefore, it is understandable that we found many papers with good and applicable ideas, but no 'silver bullet' solutions. The most comprehensive design methodologies applicable to object-oriented databases are those which were originally designed for general software development using object-oriented programming languages. There is now a consensus in the programming community that object-oriented design techniques are useful for software design no matter what type of implementation language is used [RBP+91, SM88]. In fact, object-oriented design methods are gaining acceptance as a *general* design technique. Based on our analysis, we believe that

[1] For example, some object-oriented databases are based on extended relational foundations, while others add necessary features to object-oriented programming languages such as C++. There are even those based on a functional data model, or entirely new database languages. While all of these databases share similar features, there are also enough differences to hinder development of a global development methodology. There is a trend toward standardization, however. *The Object-Oriented Database System Manifesto* [ABD+90] defines features that should be common in object-oriented databases.

object-oriented design methods such as OMT are the best tool for designing object-oriented databases.

3 Object Modeling Technique

The following brief discussion of OMT is meant to serve purely as introduction. It cannot explain important details of OMT, nor provide the reader with many illustrative examples which seem to be important for understanding. For a more complete treatment, refer to [RBP+91].

As an outline, we will begin by identifying OMT's basic strength. Next, we'll discuss its three analysis models and show how they relate to one another and to different software efforts. Then, to show how OMT is used in design, we'll walk through an example while we describe OMT methodology.

3.1 Key Idea

Although it is certainly not the only utility, the primary usefulness of OMT is for developing a concise, concrete, conceptual model of a real world problem and communicating it among a group of designers, implementors, and users. The design products of OMT capture the domain elements that describe *who, what, where, when,* and *how*[2] (and, if followed religiously, *why* also) in formal notation. By doing so, all the important information about an application and the real world it models is explicitly represented and commonly understood. The development team can manipulate the information and everyone has the same idea about what the manipulations mean.

This conceptual model of the real world is recognized as being very important. Brooks, for example, has said that the inherent complexity in software development is in the manipulation of the essence of the domain problem, rather than in the implementation of a correct conceptual model [Bro87]. By providing a clear and concise conceptual model, OMT strongly supports implementation of even the most complex applications using disparate implementation languages and tools.

[2] We've used a metaphor from the world of newspaper journalism, where editors insist that the first paragraph of copy contain all the essential information. This metaphor works because both editors and designers desire concise clarity.

ordering	indicates associations contain implicit ordering constraints
qualification	reduces multiplicity, a form of ternary association
aggregation	treated as a special association
inheritance	accompanies generalization association
constraints	explicitly included on object diagram, rather than write-up
derived data	explicitly indicated–may be objects, links, attributes
homomorphism	explicitly indicated

Table 1: Object Model Extensions of E-R Model

3.2 Three Models and Their Notation

In OMT, there are actually three different models that make up what we have called the conceptual model of an application domain. These three models are tailored to capture the structural, dynamic, and functional facets of the real-world domain. Together, they are more descriptive than any single model. However the three models naturally vary in importance with the application being developed.

The notation used to describe a design is critical to adequately representing and communicating ideas. The notation must be expressive, yet simple. To understand the choice of notation, first we must understand the underlying models.

3.2.1 Object Model

An object model is used to identify and represent important objects, attributes, and relationships between objects. The object model is an extension of the familiar entity-relationship (E-R) model, but has additional constructs, as shown in Table 1. It captures the static structure of entities in a domain. The object model tells *who* or *what* is important in a particular domain, and how these objects are related to one another.

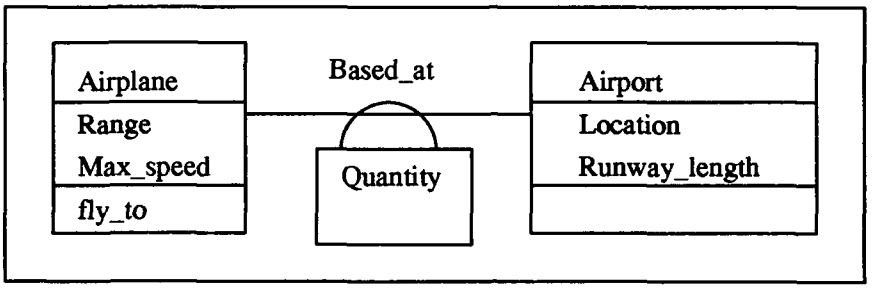


Figure 1: An Object Diagram with a Link Attribute

The object model has a corresponding object diagram which is similar to an E-R diagram. See Figure 1. Objects are depicted by boxes indicating the class name, attributes with their type and any default values, and operations with their arguments and return types. Not all of this information is required, but, if known, it can be captured on the diagram. The complete notation from [RBP+91] is shown in Figure 2.

As in other object-oriented literature, OMT makes a distinction between object classes or sets of individual objects and individual instance objects. For example, the object diagram shown in Figure 1 corresponds to classes of aircraft and airports, rather than any particular ones. Typically, we make object diagrams on classes, not instances. Considering object classes is fundamental to the concept of inheritance. Inheritance allows a concise and natural description during design and causes software re-use on implementation. Instance objects inherit class attributes and operations, i.e., those which are common to the class as a whole. Many real-world objects can be considered part of a generalization hierarchy, hence inheritance is a popular abstraction mechanism.

Relationships in OMT are called associations or links, depending on whether they describe relationships on classes or instances, respectively. Associations on an object diagram are indicated by a line drawn between boxes. The line end points have symbols to represent the cardinality of the association, an important piece of structural information. Associations can have attributes, just as in the E-R model. If present, these attributes are called 'link attributes' and are denoted by a box connected with a loop to the association. It is also possible to model an association as a class by itself. Generalization (IS-A) and aggregation (A-PART-OF) are treated as special types of associations. See Figure 2 for the special notation used.

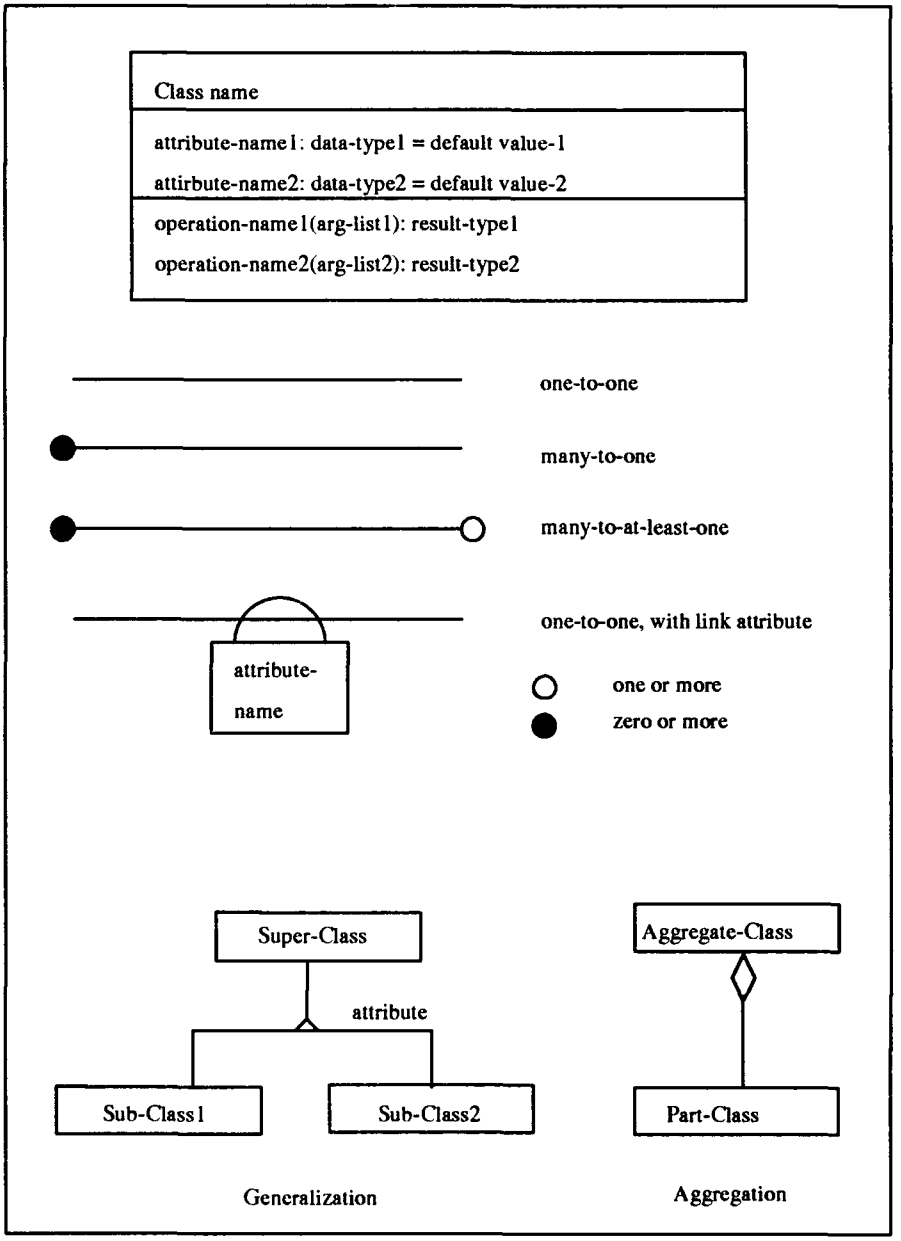


Figure 2: Object Diagram Notation

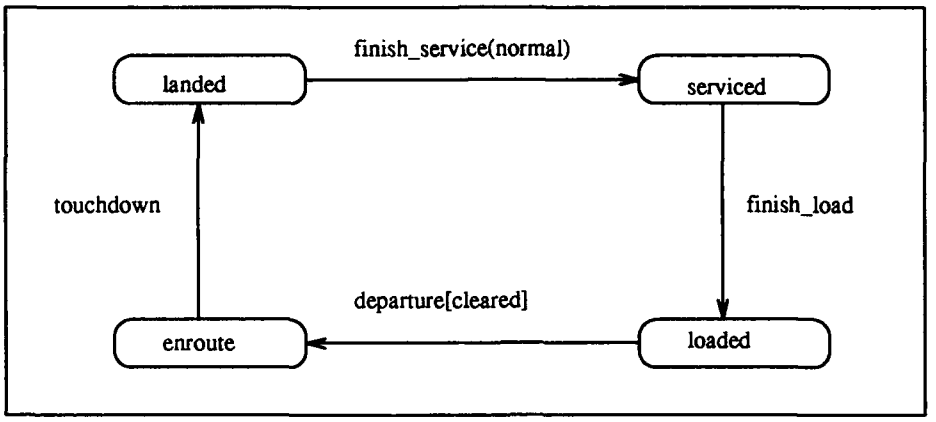


Figure 3: A State Diagram with an Event Attribute and an Event Precondition

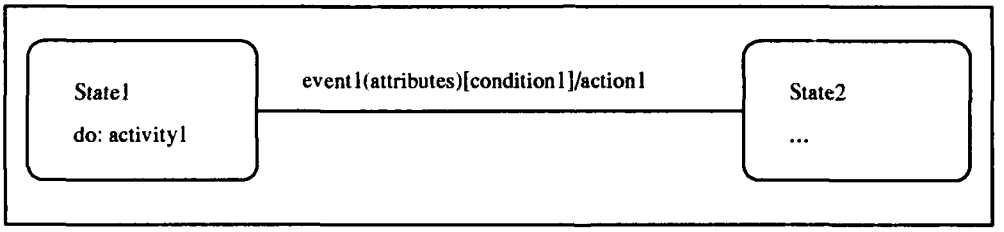


Figure 4: State Diagram Notation

3.2.2 Dynamic Model

The second type of model used in OMT represents *when* and *where* events take place and *how* the events change the state of a system of objects. The dynamic model captures state transitions and the preconditions which trigger them. The dynamic model is concerned with timing and control issues and is used to describe when and where actions occur. States correspond to changes in one or more object attribute values. To represent this information, a state diagram is used. Rounded boxes in the state diagram represent states, and the lines connecting to boxes represent events. See Figure 3. State diagrams are based on state transition diagrams. Their extended semantics include conditions on transitions, attributes of triggering events, and finite duration activities. See

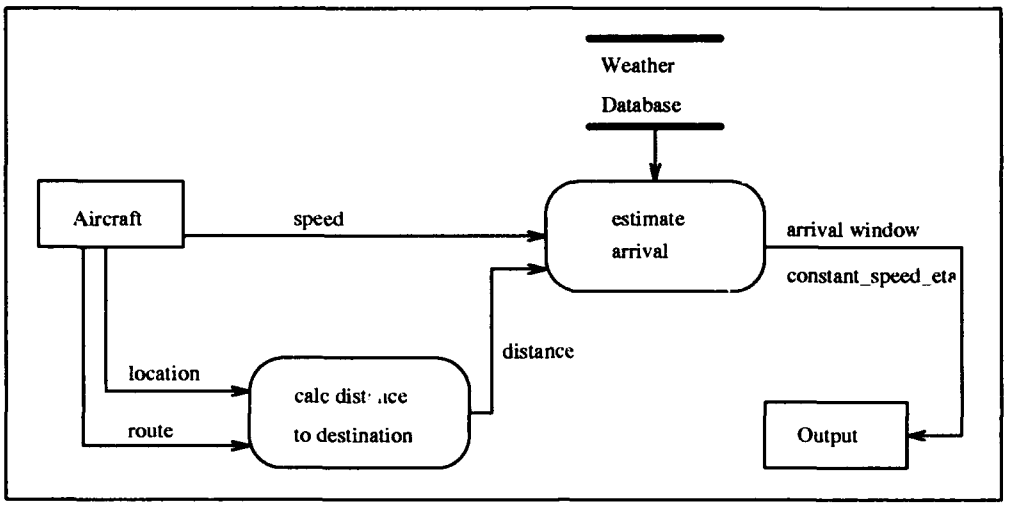


Figure 5: A Data Flow Diagram

Figure 4 for OMT's complete state diagram notation [RBP+91]. For complicated cases, the nested notation of Harel's Statecharts [Har87] is recommended to improve scalability. Sometimes it's helpful to use text scenarios and event traces to build state diagrams. We'll introduce scenarios when we discuss the OMT methodology.

3.2.3 Functional Model

Finally, OMT's functional model is used to specify *what* happens to objects and *how* their attribute values are changed as a result. The functional model specifies operations declaratively, in terms of a black-box translation function between input and output values. In addition, the functional model is the right place to specify any constraints on operations. The functional model is captured on conventional data flow diagrams. See Figures 5 and 6.

3.2.4 The Relationship Between Models

The object, dynamic, and functional models complement one another to provide a complete and descriptive overall conceptual model. The object model

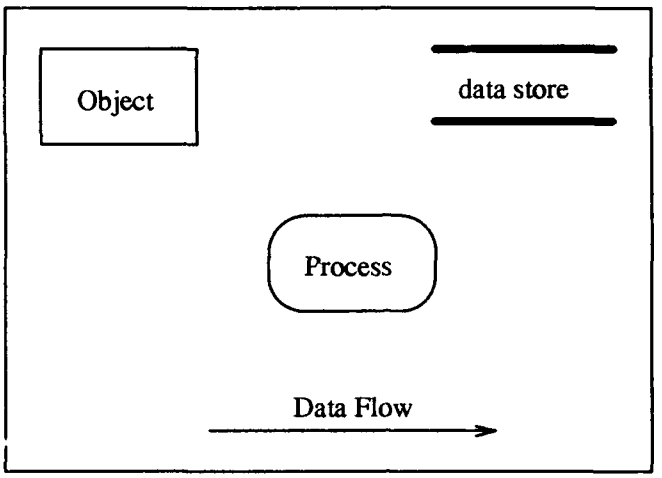


Figure 6: Data Flow Diagram Notation

captures object structure and names of important operations, but doesn't define inputs or outputs of operations or specify their control. The dynamic model uses the object states indicated by the object model and describes when object values change, but doesn't say anything about the transitions themselves. The functional model also uses objects and attributes from the object model to declaratively define operations, but it doesn't specify when changes occur, or how the changes are brought about.

3.2.5 Roles of Analysis Models in Design

The importance of the object, dynamic, and functional models depends on the nature of the software system being designed. Because databases are generally static structures that don't typically transform data, the object model is dominant for database design. Event-driven or time-sensitive applications depend heavily on the dynamic model. For example, the dynamic model is important in applications such as interactive interfaces or real-time simulations. Computation-intensive programs are typically designed using functional models. Compilers are the best example of purely functional programs.

3.3 OMT Methodology and a Design Example

3.3.1 Methodology Overview

OMT has four stages which are appropriate during different phases of development and have different end products.

1. Analysis. Developers work with domain experts to develop the basic conceptual model of the real-world application. The overall conceptual model is comprised of the object, dynamic, and functional models.

2. System Design. At a macro level, the system is partitioned and concurrency is identified. Ideally, the computer system architecture is selected and allocated based on performance requirements. General data management and control strategies are specified.

3. Object Design. Using the selected system and conceptual models, developers identify suitable data structures and algorithms for all needed objects, largely by augmenting the conceptual model.

4. Implementation. Objects and operations are implemented in a particular programming language and become an executable program.

These OMT stages are intended to be used iteratively and as needed, rather than in the cookbook fashion that our presentation might suggest. Furthermore, OMT can be applied in either a conventional top-down fashion or a rapid prototyping approach.

3.3.2 Details

We now embark on a description of how OMT is used to design a software system. To make the discussion more concrete, we will walk through the design using an example. However, the following methodology outline will be a useful roadmap along the way.

Object Modeling Technique (OMT) Methodology Outline
    Analysis
        Object Model
            Jointly construct a problem statement
            Identify objects, associations, and attributes
            Organize objects using inheritance; verify access paths
            Refine object model iteratively

Dynamic Model
- Develop scenarios of typical transactions
- Identify events and objects they impact
- Make a state diagram
- Verify model, even on odd cases

Functional Model
- Identify input and output values with data sources and sinks
- Make a data flow diagram
- Identify constraints
- State optimization requirements

System Design
- Partition the system into vertical and horizontal subsystems
- Identify inherent concurrency
- Identify needed hardware and allocate subsystems to hardware resources
- Specify a data store and management strategy
- Identify global resources and a resource control strategy
- Specify a software control mechanism
- Address initialization, termination, and error behaviors

Object Design
- Merge object, dynamic, and functional models
- Make concrete objects and operations on them
- Choose efficient algorithms and data structures
- Refine the object structure using inheritance
- Choose pointer or object representations for associations
- Allocate objects to modules

Implementation

3.3.3 Analysis

Object Model At the onset, users try to convey their needs to developers through dialog. It is helpful to write down a statement of the problem in text, even if it is not complete or error-free. For our example problem, let's design a transportation simulator for analyzing military supply problems. Our treatment of this complex problem will be necessarily abstract, but it should be useful for illustrating OMT in action. Our problem statement is shown below.

> Problem statement: We need a simulator to determine the feasibility of moving personnel and equipment from air and sea bases in the US to bases overseas. We have a large amount of data on different lift assets (mainly aircraft and ships), the available airports and seaports, and the relevant characteristics of the equipment to be moved. This data includes, among many other things, lift capacity, speed, range, and current location, port capacity and location, and size and weight of equipment. We also have a schedule of movements to make. The schedule includes a group of personnel or equipment, the destination port, and the required delivery date. We need to determine whether we can transport everything on or before dates in the movement schedule.

Object class candidates are the nouns in the problem statement, although there may be other classes implied. Many nouns will not give rise to classes, because they would be redundant, irrelevant, vague, or play some other part in the design, such being an attribute or operation. For example, air bases are also airports, so these two class are redundant. Because our focus is on transportation, we use **airport** to describe an air base. By similarly culling the other candidates, we obtain classes such as **lift_asset**, **airport**, **seaport**, **personnel**, **equipment**, and **movement_schedule**. See Figure 7.

The problem statement has no explicit associations between these objects. However, we can quickly see the need for some pieces of information which are best modeled by associations. For example, personal and equipment are **located_at** certain seaports. For that matter, lift assets are also **based_at** an airport or seaport. In fact, **movement_schedule** is actually an association between **personnel** or **equipment** and **airport** or **seaport**. Modeled in this way, **date** is a link attribute. See Figure 8.

The problem statement lists some object attributes, although there would be many more to consider if our treatment of this example were more detailed. For now, let's stick with the attributes explicitly mentioned. Therefore, a **lift_asset** has a **capacity**, **speed**, **range**, and **location**. An **airport** or **seaport** has a **capacity** and **location**. **Equipment** and **personnel** have a **size** and **weight**. We can always add attributes if we discover a need for them later.

At this point, we can begin to see that some of our classes can be abstracted to more general classes. For example, we have already abstracted from aircraft and ships to **lift_asset**. We can do the same thing with **airport** and **seaport**, obtaining **port**. Furthermore, from a transportation standpoint, **personnel** and

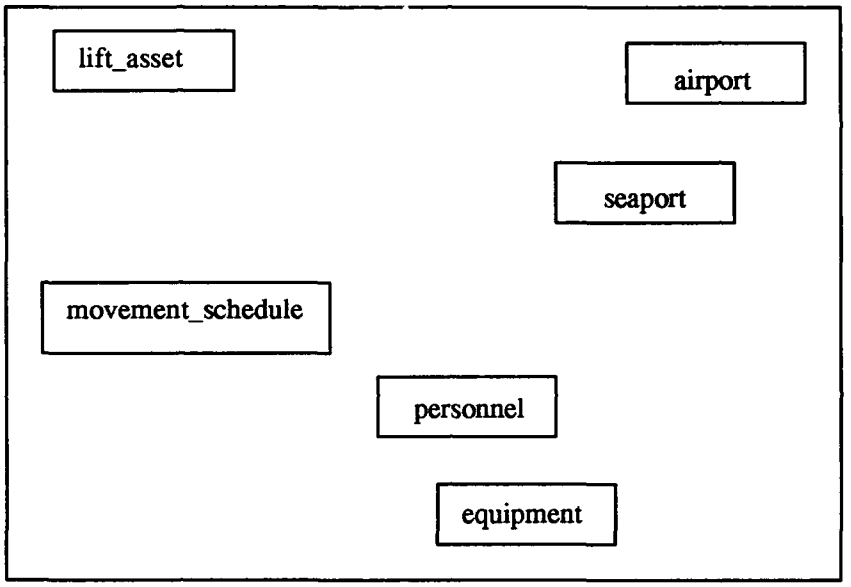


Figure 7: Transportation Object Classes

`equipment` share some of the same characteristics, and we can generalize those two classes into the `load` class.

It's also a good time to make sure that a data path exists for all anticipated operations, and that unique values, when they occur, can be found. In fact, we should iteratively go over our model to make sure to we haven't missed any objects or associations, misplaced any attributes or associations, or included any unnecessary classes or associations. If the model is large, it will also behoove us to group related classes into modules. For our example, we can assume that our current object model is sufficient. Figure 9 is our final object diagram.

Dynamic Model We don't see any stated timing or control requirements in the problem statement. This is not unusual for a problem like this batch simulation. We would expect more dynamic requirements if our problem required an interactive interface. For our problem, the dynamics depend somewhat on the algorithm we choose to for the simulation. We start building the dynamic model by building scenarios of typical interactions. By building scenarios, unstated preconditions and key events become clear. Here's an example of a typical

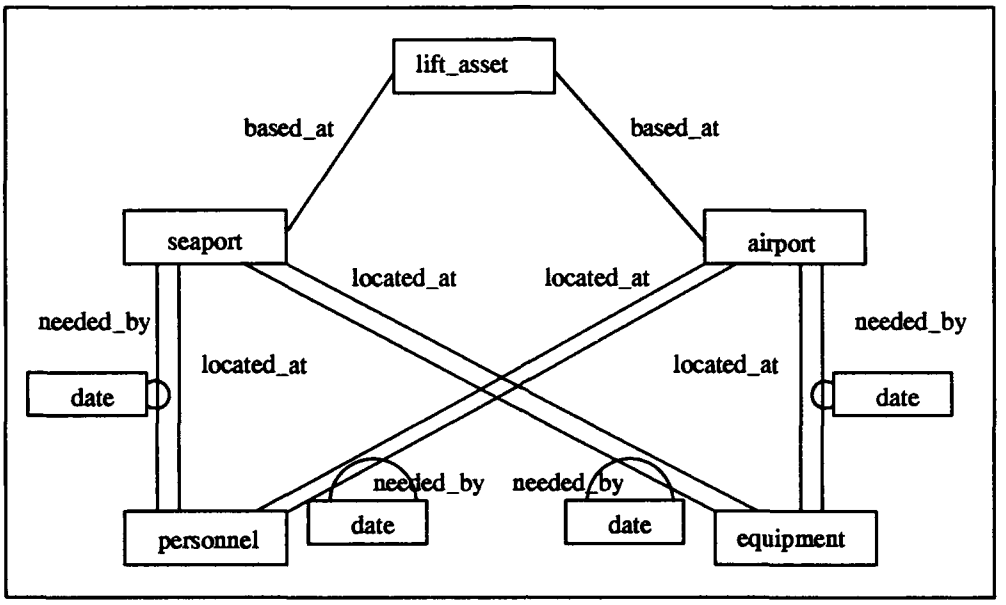


Figure 8: Some Associations Between Classes

scenario for airlifting personnel. We could make a similar scenario for sealift, or we could make an abstract example using our abstract classes `lift_asset`, `port`, and `load`.

Scenario for airlifting personnel:

A passenger aircraft becomes available
A group of people at an airport exist on the movement schedule
The group of people will fit on the aircraft and should be moved
The aircraft flies to the location of the group of people
The aircraft is serviced
The people board the aircraft
The aircraft flies to the destination airport
The people leave the aircraft
The aircraft is serviced
The movement schedule is updated
The aircraft becomes available

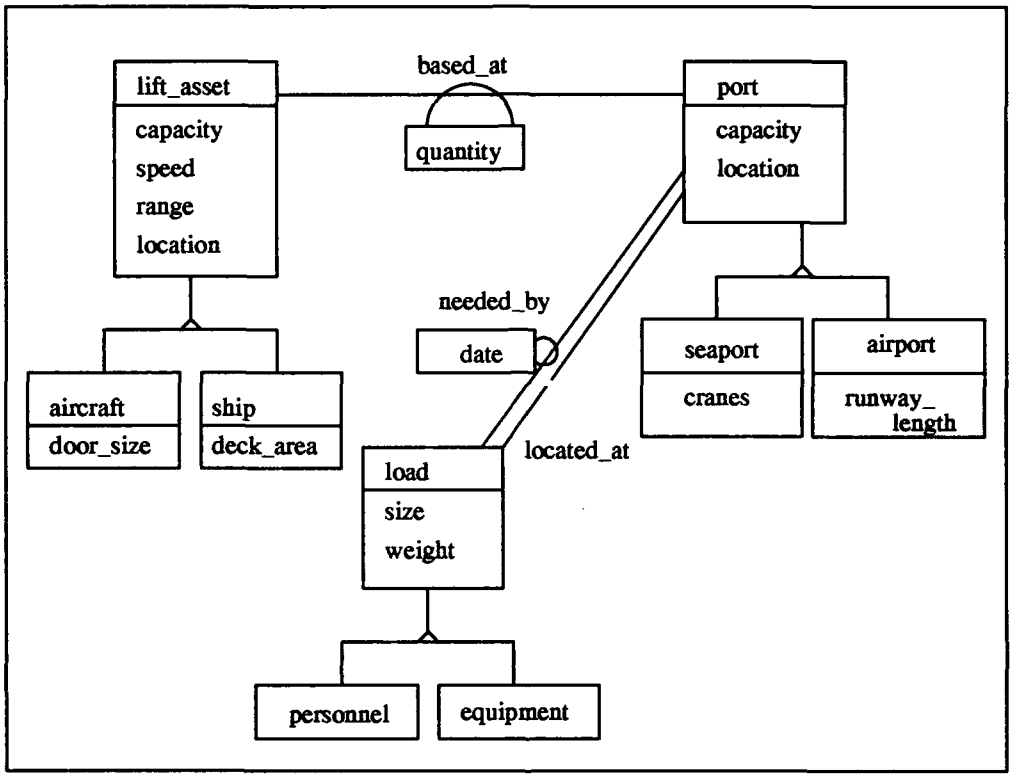


Figure 9: Transportation Simulator Object Diagram

Some of the events in our scenario are hard to visualize or give an algorithm for. For example, it is clear that somehow, we must make a decision that allocates a particular aircraft to a particular group of people on the movement schedule. At this point (and throughout the analysis stage), we should avoid worrying about *how* to do things. Rather than finding algorithms for an operation, we need to focus on specifying the events that trigger it.

Figure 10 is a state diagram for this scenario. Notice that we have identified preconditions on some triggering events. To complete the dynamic model, we should make state diagrams for all significant interactions between objects. We should also check states to make sure they correspond to objects. By doing so, we may find that we need to add attributes to our object model. In our example

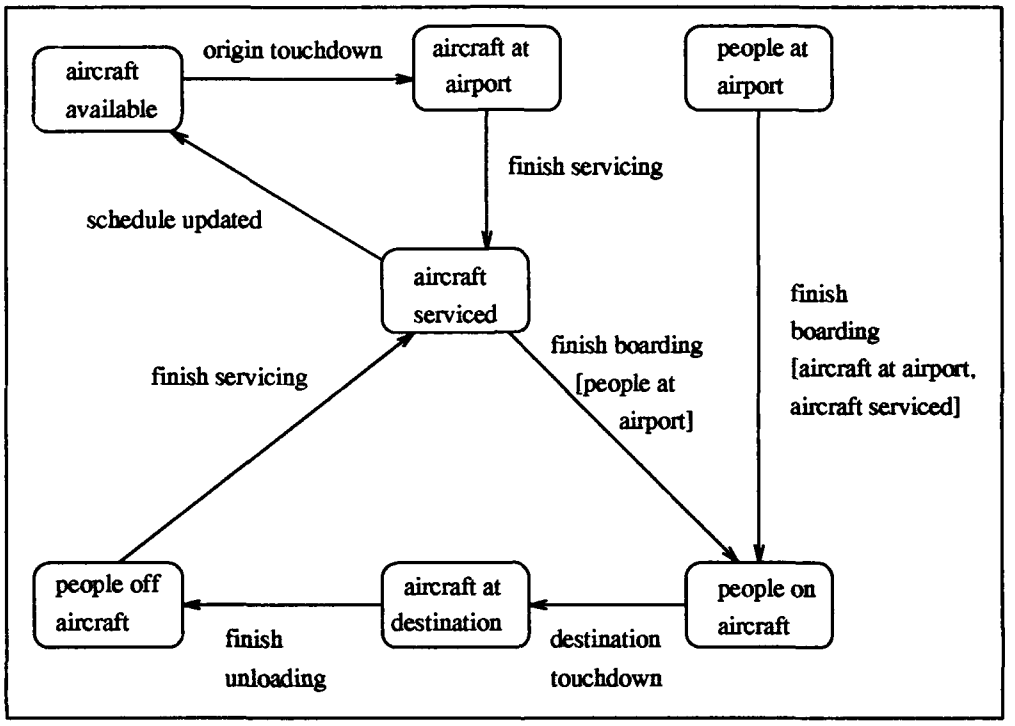


Figure 10: An Airlift State Diagram

problem, we have no `aircraft` attribute to denote the state of service, so we should add this attribute if we are modeling to that level of detail. For brevity, let's assume our dynamic model is complete.

Functional Model Having modeled the structure and dynamics of our problem, we need to shift our focus on specifying the processes that produce the output we need from the attribute values we have. When doing so, we need to pay special attention to any constraints that are inherent in the domain and to performance goals. Without considering inter-object constraints and system optimization criteria, our specification will be dangerously incomplete.

From the problem statement, we can infer that the desired output is a report which describes usages of lift assets and ports and a time line for delivery of loads. Any late deliveries should be indicated. From this description, we can

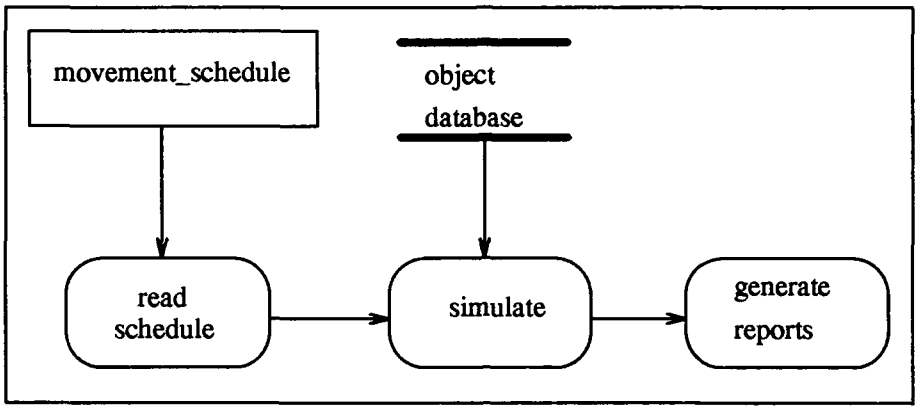


Figure 11: A High-level Data Flow Diagram for the Transportation Simulator

make a high-level data flow diagram which can be recursively extended to add detail. See Figure 11. For example, Figure 12 is a data flow diagram for the calculation of the simulation itself.

Processes on a data flow diagram should be further specified if they are not apparent from the process name and input/output values. Processes may be specified using pre- and post-conditions, mathematical formulas or equations, pseudo-code, or natural language.

How well does our system need to work and how should we state performance requirements? Performance is often a hot topic of debate among simulation users because standard metrics are not available. No simulation is perfect, but we want our simulation to give us 'reasonable' results. As developers, we depend on domain experts to define 'reasonable,' and it is our responsibility to obtain this information and capture it in our models. For example, perhaps our model must predict the average delivery date to within 3 days, given no asset or port failure. A more common criteria is simulation run-time. For example, we might specify that run-time is to be optimized, and a run-time of more that 1 hour is unacceptable.

3.3.4 Iterating the Conceptual Model

Our brief treatment of the transportation simulator highlights the need to continually refine the conceptual model. Resources spent to make a clean, correct,

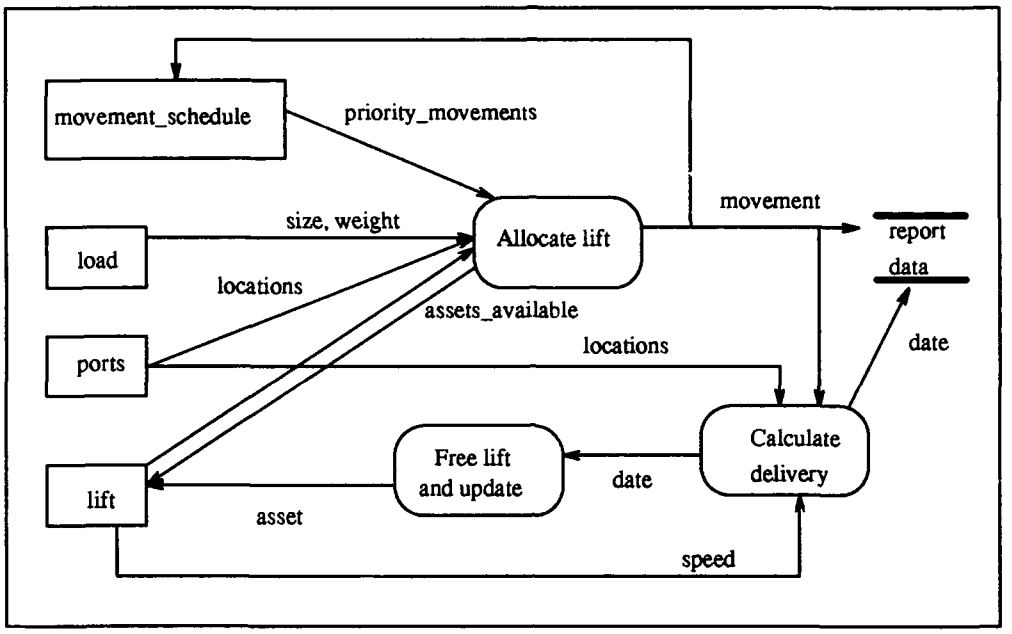


Figure 12: A Detailed Data Flow Diagram

conceptual model return the highest development dividends. Later design and implementation depend heavily on a thorough analysis and understanding.

3.3.5 System Design

Once we have done a thorough problem analysis and have a good conceptual model, it's time to consider system design. During this stage, we try to break up the system into manageable chunks and develop an architecture for storing data and executing processes.

Software systems are often partitioned using vertical slices and horizontal layers. For our example, a general partitioning scheme is shown in Figure 13, which we can recursively extend to show more detail. We can also see from our dynamic model, that some objects are inherently concurrent. For example, it is clear that the location of the aircraft before people have boarded is independent of the location of the people. However, once the people board the airplane, events which are significant to the airplane also impact the people. Identify-

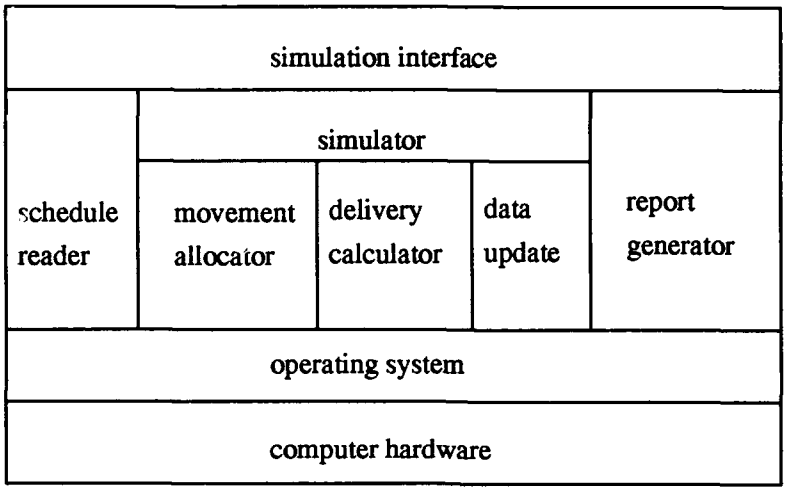


Figure 13: A Block Diagram for a Transportation Simulator

ing concurrency allows us to assign different parts of the system to different processes and processors, if we so desire.

Too often, decisions about supporting hardware and software are made without regard to design requirements, perhaps even prior to any consideration of the design. However, a better approach is to identify architectural requirements based on desired system performance. Even for developers who are constrained to use available computer systems, it's wise to document the ideal system base.

Historically, systems have been partitioned into data and operations. A data management scheme is normally required. Two obvious choices are file-based data stores and databases. For our example application, it seems wise to choose a database because we can expect that the data will be used by many users at different levels of detail for multiple applications. Using a database will provide a consistent interface as well as the normal database features which include transactions control, integrity, multi-user access, etc. With a file-based system, we would have to implement all of these services and make data structures for the sizable data inherent in our domain. Having chosen a data manager, we now need to consider an overall control scheme for operations. If we want to build an interruptible simulator, we probably want an event-driven scheme at a high level, and a procedure-driven scheme at lower levels. A batch simulator can be controlled entirely by a procedure-driven scheme.

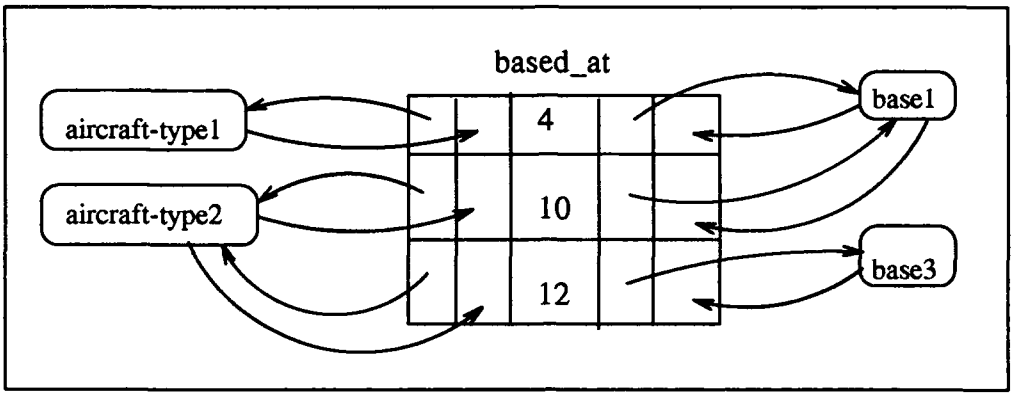


Figure 14: Implementation of an Bidirectional Association with a Link Attribute

3.3.6 Object Design

During object design, the three analysis models are merged and operations on concrete objects are identified. Efficient algorithms and data structures are chosen. A control strategy is established. Object structure is refined to maximize benefits from inheritance and associations are specified as objects or pointers. Finally, objects and associations are allocated into modules to promote modularity and re-use.

Early on in our research, we analyzed different data structures in terms of performance. Consider for example, the object diagram shown in Figure 1 which describes the objects **airplane** and **airport**, along with the **based_at** association between them. Because **based_at** has **quantity** as a link attribute, we will need to model the association as an object as shown in Figure 14. An alternative way of representing the number of aircraft **based_at** a particular airport is to use an attribute of **airport** which might have a linked list of pointers to and numbers of different aircraft in the **aircraft** class. See Figure 15. The alternative representation is only slightly more efficient, it turns out, and it can result in worse storage requirements. Furthermore, it reduces the clarity of the model. By analyzing similar examples, we have concluded that alternative data structures should be avoided in most cases. Instead, indexes or hash functions can be used to improve data access performance.

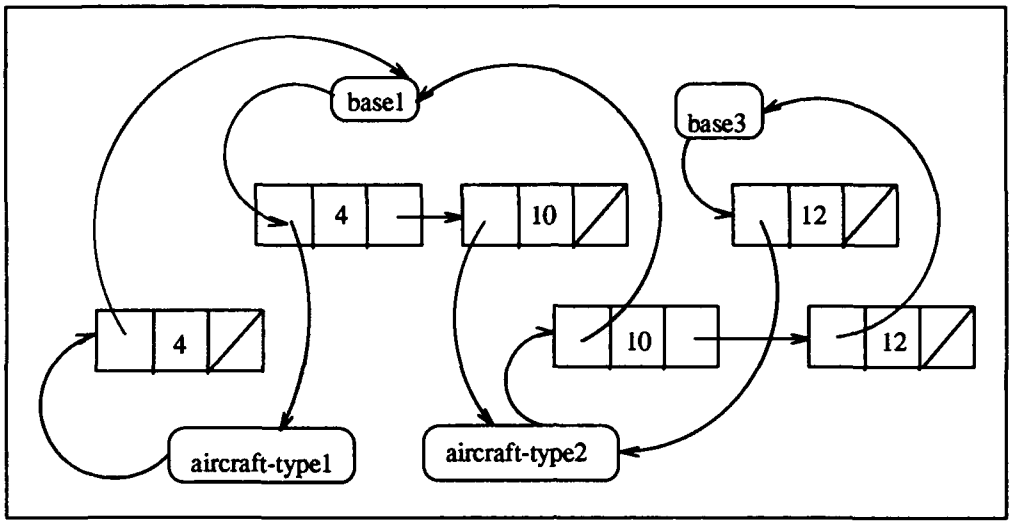


Figure 15: A Linked-List Implementation of the Same Association

For our example problem, we will retain objects as the data structure. We will implement associations without link attributes as simple pointers. Associations with attributes, such as our **based_at** example, will be implemented as objects having at least two pointer attributes which point at the objects involved in the association.

3.3.7 Implementation

Implementation, in OMT, is the same as for other design methods. We simply choose an appropriate programming language and use well-known, sound, coding practices to turn our design into the desired system. For our example, we might choose a object-oriented database written an extended programming language like persistent C++ , and then we could write the rest of the application in C++ to simplify the interface between data and operations. For example, ObjectStore is a commercial object-oriented database written in persistent C++. In ObjectStore, we would implement, say, the airplane class simply by using persistent memory which is allocated using special overloaded operator 'new' available in ObjectStore. Using persistent memory, we would define the airplane object just as we would have in normal C++.

Implementation itself is not part of design, but it is included in OMT because design and development are meant to be iterative (although we have presented OMT as if it were sequential). Some of the changes in design are stimulated by issues encountered during implementation.

4 OMT as a Database Design Tool

Database management systems are evolving from the pure relational model to other models, many of which have object-oriented features. Of the current 'object-oriented' databases, some are just extensions of the original relational model, while others are purely object-oriented or even based on the functional model. Whichever of these models become dominant (if any), previous methods of doing database design will be increasingly inappropriate. Why is this so? It is because database technology and application software technology are moving toward one another, decreasing the once valued separation between them, and thus decreasing the complexity of considering both application and database design at once. In the past, databases required very strict interfaces to application software to simplify the data model, manage complexity, and supply required services according to performance constraints. For example, we know of one transportation simulator which uses a relational database as a simple backbone for applications written in many different programming languages. The relational database is useful because it was easy to build, understandable, bulletproof, and fast enough for the job. The many issues surrounding transaction management and resource control are now much better understood; database technology can move forward and still retain virtues already won. Today's research is focused on decreasing the 'impedance mismatch[3]' between application programming languages and database data manipulation languages. On the application software side, programming languages are recognizing the need to support data modeling. Two key indicators that lend evidence to this theory are the rapid acceptance of object-oriented languages and the interest in adding persistent types to languages. In some sense, databases and application software have already merged. Some object-oriented database systems are just applications embedded in persistent C++.

[3] Impedance mismatch refers to the requirement for disparate data structures and different language syntax between application programs and the database which serves them. For current relational databases, for example, applications must download data into local data structures implemented in the application programming language, manipulate the data, then upload the data back into database tables. Moreover, the database query language has a different syntax with no control constructs or variables [Cat91, ABD+90].

The result of merging database and application software technologies is that the design methods used will also merge. As a result, designing applications and database simultaneously will become less complex. Once designers can handle the additional complexity, a single design phase is desirable, because considering the system globally often yields a better solution than if one design decision is made at a time. It will no longer be common to first design the database, and then use the stubbed-out database to build applications. Instead, the system design will be done as a single phase, and independent modules will be designed concurrently. How will this shift impact current designers? Because the shift is evolutionary in nature, the design process itself will only be re-adjusted, rather than overhauled. Database designers will work more closely with applications designers, and tasks done by each group will become more similar.

Consider the case of design using OMT, for example. Because the object model of OMT is based on and extends the E-R model, OMT is equally or more effective for design of relational, network, or hierarchical databases. Because OMT has more expressiveness than the E-R model, we can expect it to be more useful for advanced, more expressive data models and particularly suited to object-oriented database design. However, regardless of the data model used, it should be the application requirements that drive the design, rather than locally-made choices and existing frameworks as has been done previously.

5 Handling Persistence

Some object-oriented databases are based on extended programming languages such as persistent versions of C++. These database systems provide a wide-open interface to the capabilities of the underlying language, making it possible to write applications and databases in a single language. However, that means that some data will be of a transient nature, while the remainder will need to be persistent. For example, a database object may have a procedure attached to it (a method, for example, or a derived attribute). The procedure may have transient 'local' variables, but might depend as well as on some persistent attributes of the object.

From a modeling standpoint, we need to be able to identify persistence on our design documents. The simple way to do this in OMT is to add persistence to type information and indicate it on the object diagram. This is a minor addition to OMT notation which makes it applicable to our needs. Using this extended notation, persistence can be modeled even at the sub-object level, if desired.

Associations, particularly generalization, can constrain objects to be referentially persistent. For example, sub-class objects of a persistent class inherit persistence, as do instances of a class. Any bi-directional association normally requires all objects involved to either be persistent or transient. The alternative to this approach is to provide recognition and automatic dissolution of associations for disappearing transient objects. Some object-oriented databases perform persistence checking by reachability; persistence checking would be a desirable service in any automated design tool as well. In current object-oriented design methodologies, referential persistence must be checked manually.

6 Summarizing Remarks

At least a dozen different object-oriented design methodologies have been proposed over the last five years. The notation and terminology for these methods differ more than their underlying concepts. Comparisons of the different methodologies appear in several papers [BLN86, Wal92]. We would like to address some of these issues, especially as they relate to OMT.

Although there are some fundamental differences between methods, one useful comparison is the level of detail supported or demanded. There is a tradeoff between the costs and benefits of representing design information. For example, the basic entity-relationship model captures less detail than OMT's object model, but it is also simpler to construct. The design models of Shlaer and Mellor [SM88] provide another example—they are less rigorous than those of OMT, but can be built more quickly.

The level of detail supported or required by a design technique is not as important as its inherent expressiveness. Given a little experience, most design methods can be tailored to represent information which is important for a particular database or application. However, the clarity of representation can vary between models. For example, OMT's notation seems more clear than Booch's, at least to some [Wal92].

Several authors have criticized OMT's choice of the data flow diagram as notation for the functional model. The main criticism is that data flow diagrams lack clarity and do not aid in implementation [HC91, CHB92, Wal92]. One research group has suggested the use of text transition rules with pre- and post-conditions as a substitute for data flow diagrams [HC91]. To us, this is a reasonable suggestion when concrete detail is needed, but it isn't necessary in some cases.

OMT is certainly not the only proposed object-oriented design method, but it currently seems to be the most complete and consistent one. Because it subsumes the E-R model, we have concluded that it can be used to model both applications and databases. Being based on the object-oriented paradigm, OMT is particularly well suited for designs implemented in object-oriented languages or in object-oriented database systems, if persistence is annotated as necessary. The merging of design methodology reflects the larger merging of database and application software toward the object-oriented view.

References

[ABD+90] Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Sanley Zdonik. The object-oriented database system manifesto, 1990.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18, 1986.

[Bro87] Frederick P. Brooks. No silver bullet—essence and accidents of software engineering. *IEEE Computer*, April 1987.

[Cat91] R. G. G. Cattell. *Object Data Management: Object-oriented and Extended Relational Database Systems.* Addison-Wesley, 1991.

[CHB92] Derek Coleman, Fiona Hayes, and Stephen Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1), 1992.

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *The Science of Computer Programming*, 8, 1987.

[HC91] Fiona Hayes and Derek Coleman. Coherent models for object-oriented analysis. In *ACM OOPSLA '91 Conference Proceedings*, 1991.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[SM88] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Systems Analysis.* Yourdon Press, Englewood Cliffs, New Jersey, 1988.

[Wal92] Ian J. Walker. Requirements of an object-oriented design method. *Software Engineering Journal*, March 1992.